Evidence normalization in System FC

Dimitrios Vytiniotis and Simon Peyton Jones

Microsoft Research, Cambridge

Abstract

System FC is an explicitly typed language that serves as the target language for Haskell source programs. System FC is based on System F with the addition of erasable but explicit type equality proof witnesses. Equality proof witnesses are generated from type inference performed on source Haskell programs. Such witnesses may be very large objects, which causes performance degradation in later stages of compilation, and makes it hard to debug the results of type inference and subsequent program transformations. In this paper we present an equality proof simplification algorithm, implemented in GHC, which greatly reduces the size of the target System FC programs.

1998 ACM Subject Classification F.4.2 Grammars and Other Rewriting Systems

1 Introduction

A statically-typed intermediate language brings a lot of benefits to a compiler: it is free from the design trade-offs that come with source language features; types can inform optimisations; and type checking programs in the intermediate language provides a powerful consistency check on each stage of the compiler.

The Glasgow Haskell Compiler (GHC) has just such an intermediate language, which has evolved from System F to System FC [16, 20] to accommodate the source-language features of *GADTs* [6, 15, 13] and *type families* [9, 3]. The key feature that allows System FC to accomodate GADTs and type families is its use of explicit *coercions* that witness the equality of two syntactically-different types. Coercions are erased before runtime but, like types, serve as a static consistency proof that the program will not "go wrong".

In GHC, coercions are produced by a fairly complex type inference (and proof inference) algorithm that elaborates source Haskell programs into FC programs [18]. Furthermore, coercions undergo major transformations during subsequent program optimization passes. As a consequence, they can become very large, making the compiler bog down. This paper describes how we fixed the problem:

- Our main contribution is a novel coercion simplification algorithm, expressed as a rewrite system, that allows the compiler to replace a coercion with an equivalent but much smaller one (Section 4).
- Coercion simplification is important in practice. We encountered programs whose unsimplified coercion terms grow to many times the size of the actual executable terms, to the point where GHC choked and ran out of heap. When the simplifier is enabled, coercions simplify to a small fraction of their size (Section 5).
- To get these benefits, coercion simplification must take user-declared equality axioms into account, but the simplifier *must never loop* while optimizing a coercion – no matter which axioms are declared by users. Proof normalization theorems are notoriously hard, but we present such a theorem for our coercion simplification. (Section 6)

Equality proof normalization was first studied in the context of monoidal categories and we give pointers to early work in Section 7 – this work in addition addresses the simplification of open coercions containing variables and arbitrary user-declared axioms.


Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

c	$\in$	Coercion variables	
x	$\in$	Term variables	
e, u	::=	$x \mid l \mid \lambda x{:}\sigma\,.\,e \mid e\ u$	
	$\mid$	$\Lambda a{:}\eta\,.\,e \mid e\ \phi$	Type polymorphism
	$\mid$	$\lambda c{:}\tau\,.\,e \mid e\ \gamma$	Coercion abstraction/application
	$\mid$	$K \mid \textbf{case}\ e\ \textbf{of}\ \overline{p \to u}$	Constructors and case expressions
	$\mid$	$\textbf{let}\ x{:}\tau = e\ \textbf{in}\ u$	Let binding
	$\mid$	$e \triangleright \gamma$	Cast
p	::=	$K\ \overline{c{:}\tau}\ \overline{x{:}\tau}$	Patterns

Figure 1 Syntax of System FC (Terms)

Types

$\phi, \sigma, \tau, \upsilon$	::=	a	Variables
	$\mid$	H	Constants
	$\mid$	F	Type functions
	$\mid$	$\phi_1\ \phi_2$	Application
	$\mid$	$\forall a{:}\eta\,.\,\phi$	Polymorphic types

Type constants

H	::=	T	Datatypes
	$\mid$	$(\to)$	Arrow
	$\mid$	$(\sim_\#)$	Coercion

Kinds

κ, η	::=	$\star \mid \kappa \to \kappa$	
	$\mid$	$Constraint_\#$	Coercion kind

Coercion values

γ, δ	::=	c	Variables
	$\mid$	$\langle \phi \rangle$	Reflexivity
	$\mid$	$\gamma_1; \gamma_2$	Transitivity
	$\mid$	$sym\ \gamma$	Symmetry
	$\mid$	$nth\ k\ \gamma$	Injectivity
	$\mid$	$\gamma_1\ \gamma_2$	Application
	$\mid$	$C\ \overline{\gamma}$	Type family axiom
	$\mid$	$\forall a{:}\eta\,.\,\gamma$	Polym. coercion
	$\mid$	$\gamma @ \phi$	Instantiation

Figure 2 Syntax of System FC (types and coercions)

2 An overview of System FC

We begin by reviewing the role of an intermediate language. GHC desugars a rich, complex source language (Haskell) into a small, simple intermediate language. The source language, Haskell, is *implicitly typed*, and a type inference engine figures out the type of every binder and sub-expression. To make type inference feasible, Haskell embodies many somewhat ad-hoc design compromises; for example, λ-bound variables are assigned monomorphic types. By contrast, the intermediate language is simple, uniform, and *explicitly typed*. It can be typechecked by a simple, linear time algorithm. The type inference engine *elaborates* the implicitly-typed Haskell program into an explicitly-typed FC program.

To make this concrete, Figure 1 gives the syntax of System FC, the calculus implemented by GHC's intermediate language. The term language is mostly conventional, consisting of System F, together with let bindings, data constructors and case expressions. The syntax of a term encodes its typing derivation: every binder carries its type, and type abstractions $\Lambda a{:}\eta\,.\,e$ and type applications $e\,\phi$ are explicit.

The types and kinds of the language are given in Figure 2. Types include variables (a) and constants H (such as `Int` and `Maybe`), type applications (such as `Maybe Int`), and polymorphic types ($\forall a{:}\eta\,.\,\phi$). The syntax of types also includes *type functions* (or *type families* in the Haskell jargon), which are used to express type level computation. For instance the following declaration in source Haskell:

Environments

Γ, Δ	::=	$\cdot \mid \Gamma, bnd$	
bnd	::=	$a : \eta$	Type variable
	$\mid$	$c : \sigma \sim_{\#} \phi$	Coercion variable
	$\mid$	$x : \sigma$	Term variable
	$\mid$	$T : \overline{\kappa} \to \star$	Data type
	$\mid$	$K : \forall(\overline{a{:}\eta}).\overline{\tau} \to T\ \overline{a}$	Data constructor
	$\mid$	$F^n : \overline{\kappa}^n \to \kappa$	Type families (of arity n)
	$\mid$	$C\,\overline{(a{:}\eta)} : \sigma \sim_{\#} \phi$	Axioms

Notation

$T\ \overline{\tau}$	$\equiv$	$T\ \tau_1 \ldots \tau_n$
$\overline{\tau} \to \tau$	$\equiv$	$\tau_1 \to \ldots \to \tau_n \to \tau$
$\overline{\tau}^{1..n}$	$\equiv$	$\tau_1, \ldots, \tau_n$

Figure 3 Syntax of System FC (Auxiliary definitions)

```
type family F (a :: *) :: a
type instance F [a] = a
```

introduces a type function F at the level of System FC. The accompanying `instance` line asserts that any expression of type `F [a]` can be viewed as having type `a`. We shall see in Section 2.2 how this fact is expressed in FC. Finally type constants include datatype constructors (T) but also arrow ($\to$) as well as a special type constructor $\sim_{\#}$ whose role we explain in the following section. The kind language includes the familiar $\star$ and $\kappa_1 \to \kappa_2$ kinds but also a special kind called $Constraint_{\#}$ that we explain along with the $\sim_{\#}$ constructor.

The typing rules for System FC are given in Figure 4. We urge the reader to consult [16, 20] for more examples and intuition.

2.1 Coercions

The unusual feature of FC is the use of coercions. The term $e \triangleright \gamma$ is a cast, that converts a term e of type τ to one of type ϕ (rule ECast in Figure 4). The coercion γ is a *witness*, or *proof*, providing evidence that τ and ϕ are equal types – that is, γ has type $\tau \sim_{\#} \phi$. We use the symbol "$\sim_{\#}$" to denote type equality[1]. The syntax of coercions γ is given in Figure 2, and their typing rules in Figure 6. For uniformity we treat $\sim_{\#}$ as an ordinary type constructor, with kind $\kappa \to \kappa \to Constraint_{\#}$ (Figure 5).

To see casts in action, consider this Haskell program which uses GADTs:

```
data T a where                f :: T a -> [a]
   T1 :: Int -> T Int         f (T1 x) = [x+1]
   T2 :: a -> T a             f (T2 v) = [v]

                              main = f (T1 4)
```

We regard the GADT data constructor `T1` as having the type

$$\texttt{T1} : \forall a.(a \sim_{\#} \texttt{Int}) \to \texttt{Int} \to \texttt{T}\ a$$

[1] The "#" subscript is irrelevant for this paper; the interested reader may consult [19] to understand the related type equality $\sim$, and the relationship between $\sim$ and $\sim_{\#}$.

$$\boxed{\Gamma \vdash^{\mathsf{tm}} e : \tau}$$

$$\frac{(x{:}\tau) \in \Gamma}{\Gamma \vdash^{\mathsf{tm}} x : \tau}\ \textsc{EVar} \qquad \frac{(K{:}\sigma) \in \Gamma}{\Gamma \vdash^{\mathsf{tm}} K : \sigma}\ \textsc{ECon}$$

$$\frac{\Gamma, (x{:}\sigma) \vdash^{\mathsf{tm}} e : \tau \quad \Gamma \vdash^{\mathsf{ty}} \sigma : \star}{\Gamma \vdash^{\mathsf{tm}} \lambda x{:}\sigma\,.\, e : \sigma \to \tau}\ \textsc{EAbs} \qquad \frac{\Gamma \vdash^{\mathsf{tm}} e : \sigma \to \tau \quad \Gamma \vdash^{\mathsf{tm}} u : \sigma}{\Gamma \vdash^{\mathsf{tm}} e\ u : \tau}\ \textsc{EApp}$$

$$\frac{\Gamma, (c{:}\sigma) \vdash^{\mathsf{tm}} e : \tau \quad \Gamma \vdash^{\mathsf{ty}} \sigma : \mathit{Constraint}_{\#}}{\Gamma \vdash^{\mathsf{tm}} \lambda c{:}\sigma\,.\, e : \sigma \to \tau}\ \textsc{ECAbs} \qquad \frac{\Gamma \vdash^{\mathsf{tm}} e : (\sigma_1 \sim_{\#} \sigma_2) \to \tau \quad \Gamma \vdash^{\mathsf{co}} \gamma : \sigma_1 \sim_{\#} \sigma_2}{\Gamma \vdash^{\mathsf{tm}} e\ \gamma : \tau}\ \textsc{ECApp}$$

$$\frac{\Gamma, (a{:}\eta) \vdash^{\mathsf{tm}} e : \tau}{\Gamma \vdash^{\mathsf{tm}} \Lambda a{:}\eta\,.\, e : \forall a{:}\eta\,.\,\tau}\ \textsc{ETAbs} \qquad \frac{\Gamma \vdash^{\mathsf{tm}} e : \forall a{:}\eta\,.\,\tau \quad \Gamma \vdash^{\mathsf{ty}} \phi : \eta}{\Gamma \vdash^{\mathsf{tm}} e\ \phi : \tau[\phi/a]}\ \textsc{ETApp}$$

$$\frac{\Gamma, (x{:}\sigma) \vdash^{\mathsf{tm}} u : \sigma \quad \Gamma, (x{:}\sigma) \vdash^{\mathsf{tm}} e : \tau}{\Gamma \vdash^{\mathsf{tm}} \mathbf{let}\ x{:}\sigma = u\ \mathbf{in}\ e : \tau}\ \textsc{ELet} \qquad \frac{\Gamma \vdash^{\mathsf{tm}} e : \tau \quad \Gamma \vdash^{\mathsf{co}} \gamma : \tau \sim_{\#} \phi}{\Gamma \vdash^{\mathsf{tm}} e \rhd \gamma : \phi}\ \textsc{ECast}$$

$$\frac{\begin{array}{l}\Gamma \vdash^{\mathsf{tm}} e : T\ \overline{\kappa}\ \overline{\sigma} \\ \text{For each branch } K\ \overline{x{:}\tau} \to u \\ \quad (K{:}\forall(\overline{a{:}\eta_a})\,.\,\overline{\sigma_1 \sim_{\#} \sigma_2} \to \overline{\tau} \to T\ \overline{a}) \in \Gamma \\ \quad \phi_i = \tau_i[\overline{\sigma}/\overline{a}] \\ \quad \phi_{1i} = \sigma_{1i}[\overline{\sigma}/\overline{a}] \\ \quad \phi_{2i} = \sigma_{2i}[\overline{\sigma}/\overline{a}] \quad \Gamma, \overline{c{:}\phi_1 \sim_{\#} \phi_2}\ \overline{x{:}\phi} \vdash^{\mathsf{tm}} u : \sigma\end{array}}{\Gamma \vdash^{\mathsf{tm}} \mathbf{case}\ e\ \mathbf{of}\ \overline{K\ (\overline{c{:}\sigma_1 \sim_{\#} \sigma_2})\ (\overline{x{:}\tau}) \to u} : \sigma}\ \textsc{ECase}$$

Figure 4 Well-formed terms

$$\boxed{\Gamma \vdash^{\mathsf{ty}} \tau : \kappa}$$

$$\frac{(a{:}\eta) \in \Gamma}{\Gamma \vdash^{\mathsf{ty}} a : \eta}\ \textsc{TVar} \qquad \frac{(T{:}\kappa) \in \Gamma}{\Gamma \vdash^{\mathsf{ty}} T : \kappa}\ \textsc{TData} \qquad \frac{(F{:}\kappa) \in \Gamma}{\Gamma \vdash^{\mathsf{ty}} F : \kappa}\ \textsc{TFun}$$

$$\frac{\kappa_1, \kappa_2 \in \{\mathit{Constraint}_{\#}, \star\}}{\Gamma \vdash^{\mathsf{ty}} (\to) : \kappa_1 \to \kappa_2 \to \star}\ \textsc{TArr} \qquad \frac{}{\Gamma \vdash^{\mathsf{ty}} (\sim_{\#}) : \kappa \to \kappa \to \mathit{Constraint}_{\#}}\ \textsc{TEqPred}$$

$$\frac{\Gamma \vdash^{\mathsf{ty}} \phi_1 : \kappa_1 \to \kappa_2 \quad \Gamma \vdash^{\mathsf{ty}} \phi_2 : \kappa_1}{\Gamma \vdash^{\mathsf{ty}} \phi_1\ \phi_2 : \kappa_2}\ \textsc{TApp} \qquad \frac{\Gamma, (a{:}\eta) \vdash^{\mathsf{ty}} \tau : \star}{\Gamma \vdash^{\mathsf{ty}} \forall a{:}\eta\,.\,\tau : \star}\ \textsc{TAll}$$

Figure 5 Well-formed types

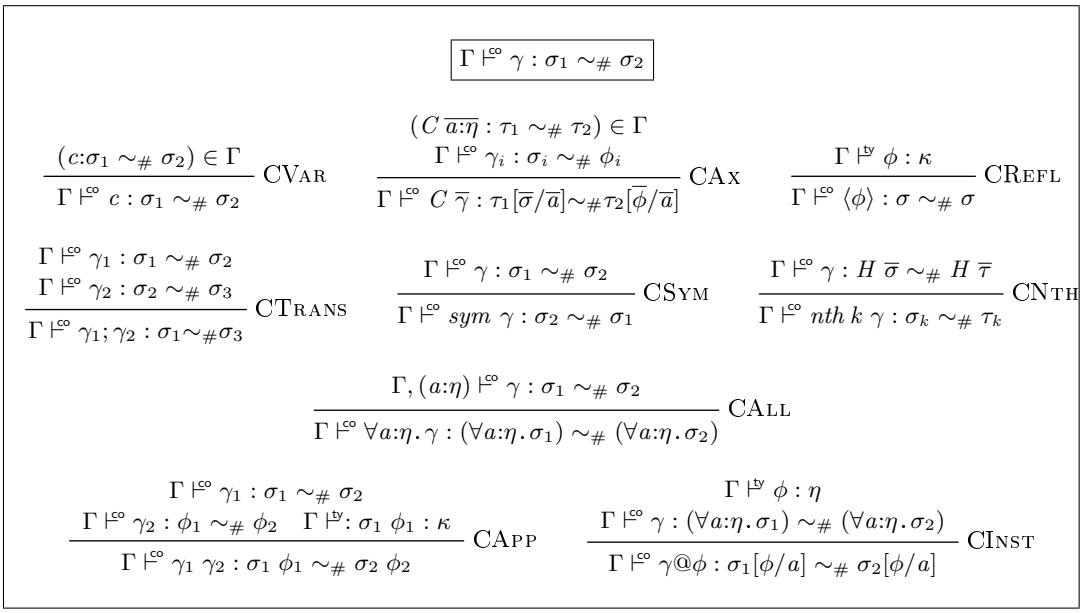

Figure 6 Well-formed coercions

So in FC, `T1` takes three arguments: a type argument to instantiate a, a coercion witnessing the equivalence of a and `Int`, and a value of type `Int`. Here is the FC elaboration of `main`:

```
main = f Int (T1 Int <Int> 4)
```

The coercion argument has kind (`Int` $\sim_\#$ `Int`), for which the evidence is just $\langle$`Int`$\rangle$ (reflexivity). Similarly, pattern-matching on `T1` binds two variables: a coercion variable, and a term variable. Here is the FC elaboration of function `f`:

```
f = /\(a:*). \(x:T a).
    case x of
      T1 (c:a ~# Int) (n:Int) -> (Cons (n+1) Nil) |> sym [c]
      T2 (v:a)                -> Cons v Nil
```

The cast converts the type of the result from `[Int]` to `[a]`. The coercion *sym* $[c]$ is evidence for (or a proof of) the equality of these types, using coercion `c`, of type (`a` $\sim_\#$ `Int`).

2.2 Typing coercions

Figure 6 gives the typing rules for coercions. The rules include unsurprising cases for reflexivity (CRefl), symmetry (CSym), and transitivity (CTrans). Rules CAll and CApp allow us to construct coercions on more complex types from coercions on simpler types. Rule CInst instantiates a coercion between two $\forall$-types, to get a coercion between two instantiated types. Rule CVar allows us to use a coercion that has been introduced to the context by a coercion abstraction ($\lambda c{:}\tau \sim_\# \phi.\, e$), or a pattern match against a GADT (as in the example above).

Rule CAx refers to instantiations of *axioms*. In GHC, axioms can arise as a result of *newtype* or *type family* declarations. Consider the following code:

```
newtype N a = MkN (a -> Int)
```

```
type family F (x :: *) :: *
type instance F [a]  = a
type instance F Bool = Char
```

N is a *newtype* (part of the original Haskell 98 definition), and is desugared to the following FC coercion axiom:

$$C_N\ a \quad : \quad N\ a \sim_\# a \to \texttt{Int}$$

which provides evidence of the equality of types $(N\ a)$ and $(a \to \texttt{Int})$.

In the above Haskell code, F is a *type family* [4, 3], and the two `type instance` declarations above introduce two FC coercion axioms:

$$\begin{array}{rcl} C_1\ a & : & F\ [a] \sim_\# a \\ C_2 & : & F\ \texttt{Bool} \sim_\# \texttt{Char} \end{array}$$

Rule CAx describes how these axioms may be used to create coercions. In this particular example, if we have $\gamma : \tau \sim_\# \sigma$, then we can prove that $C_1\ \gamma : F\ [\tau] \sim_\# \sigma$. Using such coercions we can get, for example, that $(3 \triangleright \mathit{sym}\,(C_1\ \langle \texttt{Int} \rangle)) : F\ [\texttt{Int}]$.

Axioms always appear saturated in System FC, hence the syntax $C\,\overline{\gamma}$ in Figure 2.

3 The problem with large coercions

System FC terms arise as the result of elaboration of source language terms, through type inference. Type inference typically relies on a *constraint solver* [18] which produces System FC witnesses of equality (coercions), that in turn decorate the elaborated term. The constraint solver is not typically concerned with producing small or readable witnesses; indeed GHC's constraint solver can produce large and complex coercions. These complex coercions can make the elaborated term practically impossible to understand and debug.

Moreover, GHC's optimiser transforms well-typed FC terms. Insofar as these transformations involve coercions, the coercions *themselves* may need to be transformed. If you think of the coercions as little proofs that fragments of the program are well-typed, then the optimiser must maintain the proofs as it transforms the terms.

3.1 How big coercions arise

The trouble is that *term-level optimisation tends to make coercions bigger*. The full details of these transformations are given in the so called *push* rules in our previous work [20], but we illustrate them here with an example. Consider this term:

$$(\lambda x.e \triangleright \gamma)\ a$$

where

$$\begin{array}{rcl} \gamma & : & (\sigma_1 \to \tau_1) \sim_\# (\sigma_2 \to \tau_2) \\ a & : & \sigma_2 \end{array}$$

We would like to perform the beta reduction, but the cast is getting in the way. No matter! We can transform thus:

$$\begin{array}{rcl} & & (\lambda x.e \triangleright \gamma)\ a \\ = & & ((\lambda x.e)\ (a \triangleright \mathit{sym}\,(\mathit{nth}\,0\,\gamma))) \triangleright \mathit{nth}\,1\,\gamma \end{array}$$

From the coercion γ we have derived two coercions whose syntactic form is larger, but whose types are smaller:

$$\begin{array}{rcl} \gamma & : & (\sigma_1 \to \tau_1) \sim_\# (\sigma_2 \to \tau_2) \\ sym\,(nth\,0\,\gamma) & : & \sigma_2 \sim_\# \sigma_1 \\ nth\,1\,\gamma & : & \tau_1 \sim_\# \tau_2 \end{array}$$

Here we make use of the coercion combinators *sym*, which reverses the sense of the proof; and *nth* i, which from a proof of $T\,\overline{\sigma} \sim_\# T\,\overline{\tau}$ gives a proof of $\sigma_i \sim_\# \tau_i$. Finally, we use the derived coercions to cast the argument and result of the function separately. Now the lambda is applied directly to an argument (without a cast in the way), so β-reduction can proceed as desired. Since β-reduction is absolutely crucial to the optimiser, this ability to "push coercions out of the way" is fundamental. Without it, the optimiser is hopelessly compromised.

A similar situation arises with `case` expressions:

$$\texttt{case}\,(K\ e_1 \triangleright \gamma)\,\texttt{of}\,\{\ldots; K\,x \to e_2; \ldots\}$$

where K is a data constructor. Here we want to simplify the `case` expression, by picking the correct alternative $K\,x \to e_2$, and substituting e_1 for x. Again the coercion gets in the way, but again it is possible to push the coercion out of way.

3.2 How coercions can be simplified

Our plan is to simplify complicated coercion terms into simpler ones, using rewriting. Here are some obvious rewrites we might think of immediately:

$$\begin{array}{rcll} sym\,(sym\ \gamma) & \rightsquigarrow & \gamma & \\ \gamma; sym\ \gamma & \rightsquigarrow & \langle\tau\rangle & \text{if}\,\gamma : \tau \sim_\# \phi \end{array}$$

But ther are much more complicated rewrites to consider. Consider these coercions, where C_N is the axiom generated by the newtype coercion in Section 2.2:

$$\begin{array}{rcl} \gamma_1 & : & \tau_1 \sim_\# \tau_2 \\ \gamma_2 = sym\,(C_N\,\langle\tau_1\rangle) & : & (\tau_1 \to \texttt{Int}) \sim_\# (N\,\tau_1) \\ \gamma_3 = N\,\langle\gamma_1\rangle & : & (N\,\tau_1) \sim_\# (N\,\tau_2) \\ \gamma_4 = C_N\,\langle\tau_2\rangle & : & (N\,\tau_2) \sim_\# (\tau_2 \to \texttt{Int}) \\ \\ \gamma_5 = \gamma_2;\gamma_3;\gamma_4 & : & (\tau_1 \to \texttt{Int}) \sim_\# (\tau_2 \to \texttt{Int}) \end{array}$$

Here γ_2 takes a function, and wraps it in the newtype; then γ_3 coerces that newtype from $N\,\tau_1$ to $N\,\tau_2$; and γ_4 unwraps the newtype. Composing the three gives a rather large, complicated coercion $\gamma_2;\gamma_3;\gamma_4$. *But its type is pretty simple*, and indeed the coercion $\gamma_1 \to \langle\texttt{Int}\rangle$ is a much simpler witness of the same equality. The rewrite system we present shortly will rewrite the former to the latter.

Finally, here is an actual example taken from a real program compiled by GHC (don't look at the details!):

$$\begin{array}{rl} & \texttt{Mut}\,\langle v\rangle\,(sym\,(C_{StateT}\,\langle s\rangle))\,\langle a\rangle \\ & ; sym\,(nth\,0\,((\forall wtb.\,\texttt{Mut}\,\langle w\rangle\,(sym\,(C_{StateT}\,\langle t\rangle))\,\langle b\rangle \to \langle\texttt{ST}\,t\,(w\ b)\rangle)@v@s@a) \\ \rightsquigarrow & \langle\texttt{Mut}\ v\ s\ a\rangle \end{array}$$

As you can see, the shrinkage in coercion size can be dramatic.

4 Coercion simplification

We now proceed to the details of our coercion simplification algorithm. We note that the design of the algorithm is guided by empirical evidence of its effectiveness on actual programs and that other choices might be possible. Nevertheless, we formally study the properties of this algorithm, namely we will show that it preserves validity of coercions and terminates – even when the rewrite system induced by the axioms is not strongly normalizing.

4.1 Simplification rules

Coercion simplification is given as a non-deterministic relation in Figure 7 and Figure 8 In these two figures we use some syntactic conventions: Namely, for sequences of coercions $\overline{\gamma}_1$ and $\overline{\gamma}_2$, we write $\overline{\gamma_1;\gamma_2}$ for the sequence of pointwise transitive compositions and $sym\ \overline{\gamma}_1$ for pointwise application of symmetry. We write $nontriv(\gamma)$ iff γ *contains* some variable c or axiom application $C\ \overline{\gamma}$.

We define coercion evaluation contexts, $\mathcal{G}$, as coercion terms with holes inside them. The syntax of $\mathcal{G}$ allows us to rewrite anywhere inside a coercion. The main coercion evaluation rule is CoEval. If we are given a coercion γ, we first decompose it to some evaluation context $\mathcal{G}$ with γ_1 in its hole. Rule CoEval works up to associativity of transitive composition; for example, we will allow the term $(\gamma_1;\gamma_2;);\gamma_3$ to be written as $\mathcal{G}[\gamma_2;\gamma_3]$ where $\mathcal{G} = \gamma_1;\Box$. This treatment of transitivity is extremely convenient, but we must be careful to ensure that our argument for termination remains robust under associativity (Section 6). Once we have figured out a decomposition $\mathcal{G}[\gamma_1]$, CoEval performs a single step of rewriting $\Delta \vdash \gamma_1 \rightsquigarrow \gamma_2$ and simply return $\mathcal{G}[\gamma_2]$. Since we are allowed to rewrite coercions under a type environment ($\forall a{:}\eta.\mathcal{G}$ is a valid coercion evaluation context), Δ (somewhat informally) enumerates the type variables bound by $\mathcal{G}$. For instance we should be allowed to rewrite $\forall a{:}\eta.\gamma_1$ to $\forall a{:}\eta.\gamma_2$. This can happen if $(a{:}\eta)| - \gamma_1 \rightsquigarrow \gamma_2$. The precondition $\Delta \vDash^{\infty} \gamma_1 : \sigma \sim_{\#} \phi$ of rule CoEval ensures that this context corresponds to the decomposition of γ into a context and γ_1. Moreover, the Δ is passed on to the $\rightsquigarrow$ relation, since some of the rules of the $\rightsquigarrow$ relation that we will present later may have to consult the context Δ to establish preconditions for rewriting.

The soundness property for the $\longrightarrow$ relation is given by the following theorem.

▶ **Theorem 1** (Coercion subject reduction). *If* $\vDash^{\infty} \gamma_1 : \sigma \sim_{\#} \phi$ *and* $\gamma_1 \longrightarrow \gamma_2$ *then* $\vDash^{\infty} \gamma_2 : \sigma \sim_{\#} \phi$.

The rewriting judgement $\Delta \vdash \gamma_1 \rightsquigarrow \gamma_2$ satisfies a similar property.

▶ **Lemma 2.** *If* $\Delta \vDash^{\infty} \gamma_1 : \sigma \sim_{\#} \phi$ *and* $\Delta \vdash \gamma_1 \rightsquigarrow \gamma_2$ *then* $\Delta \vDash^{\infty} \gamma_2 : \sigma \sim_{\#} \phi$.

To explain coercion simplification, we now present the reaction rules for the $\rightsquigarrow$ relation, organized in several groups.

4.1.1 Pulling reflexivity up

Rules ReflApp, ReflAll, ReflElimL, and ReflElimR, deal with uses of reflexivity. Rules ReflApp and ReflAll "swallow" constructors from the coercion language (coercion application, and quantification respectively) into the type language (type application, and quantification respectively). Hence they pull reflexivity as high as possible in the tree structure of a coercion term. Rules ReflElimL and ReflElimR simply eliminate reflexivity uses that are composed with other coercions.

Coercion evaluation contexts $\mathcal{G} ::= \Box \mid \mathcal{G}\ \gamma \mid \gamma\ \mathcal{G} \mid C\ \overline{\gamma}_1 \mathcal{G} \overline{\gamma}_2 \mid sym\ \mathcal{G} \mid \forall a{:}\eta.\mathcal{G} \mid \mathcal{G}@\tau \mid \mathcal{G};\gamma \mid \gamma;\mathcal{G}$

$$\frac{\gamma \cong \mathcal{G}[\gamma_1] \text{ modulo associativity of } (;) \quad \Delta \vdash^{\text{co}} \gamma_1 : \sigma \sim_{\#} \phi \quad \Delta \vdash \gamma_1 \rightsquigarrow \gamma_2}{\gamma \longrightarrow \mathcal{G}[\gamma_2]}\ \textsc{CoEval}$$

$\boxed{\Delta \vdash \gamma_1 \rightsquigarrow \gamma_2}$

Reflexivity rules

RefLApp	$\Delta \vdash \langle\phi_1\rangle\ \langle\phi_2\rangle$	$\rightsquigarrow$	$\langle\phi_1\ \phi_2\rangle$
RefLAll	$\Delta \vdash \forall a{:}\eta.\langle\phi\rangle$	$\rightsquigarrow$	$\langle\forall a{:}\eta.\phi\rangle$
RefLElimL	$\Delta \vdash \langle\phi\rangle;\gamma$	$\rightsquigarrow$	γ
RefLElimR	$\Delta \vdash \gamma;\langle\phi\rangle$	$\rightsquigarrow$	γ

Eta rules

EtaAllL	$\Delta \vdash ((\forall a{:}\eta.\gamma_1);\gamma_2)@\phi$	$\rightsquigarrow$	$\gamma_1[\phi/a];(\gamma_2@\phi)$
EtaAllR	$\Delta \vdash (\gamma_1;(\forall a{:}\eta.\gamma_2))@\phi$	$\rightsquigarrow$	$\gamma_1@\phi;\gamma_2[\phi/a]$
EtaNthL	$\Delta \vdash nth\ k\ (\langle H\ \overline{\tau}^{1..\ell}\rangle\ \overline{\gamma};\gamma)$	$\rightsquigarrow$	$\begin{cases} nth\ k\ \gamma & \text{if } k \le \ell \\ \gamma_{k-\ell};nth\ k\ \gamma & \text{otherwise} \end{cases}$
EtaNthR	$\Delta \vdash nth\ k\ (\gamma;\langle H\ \overline{\tau}^{1..\ell}\rangle\ \overline{\gamma})$	$\rightsquigarrow$	$\begin{cases} nth\ k\ \gamma & \text{if } k \le \ell \\ nth\ k\ \gamma;\gamma_{k-\ell} & \text{otherwise} \end{cases}$

Symmetry rules

SymRefl	$\Delta \vdash sym\ \langle\phi\rangle$	$\rightsquigarrow$	$\langle\phi\rangle$
SymAll	$\Delta \vdash sym\ (\forall a{:}\eta.\gamma)$	$\rightsquigarrow$	$\forall a{:}\eta.\ sym\ \gamma$
SymApp	$\Delta \vdash sym\ (\gamma_1\ \gamma_2)$	$\rightsquigarrow$	$(sym\ \gamma_1)\ (sym\ \gamma_2)$
SymTrans	$\Delta \vdash sym\ (\gamma_1;\gamma_2)$	$\rightsquigarrow$	$(sym\ \gamma_2);(sym\ \gamma_1)$
SymSym	$\Delta \vdash sym\ (sym\ \gamma)$	$\rightsquigarrow$	γ

Reduction rules

RedNth	$\Delta \vdash nth\ k\ (\langle H\ \overline{\tau}^{1..\ell}\rangle\ \overline{\gamma})$	$\rightsquigarrow$	$\begin{cases} \langle\tau_k\rangle & \text{if } k \le \ell \\ \gamma_{k-\ell} & \text{otherwise} \end{cases}$
RedInstCo	$\Delta \vdash (\forall a{:}\eta.\gamma)@\phi$	$\rightsquigarrow$	$\gamma[\phi/a]$
RedInstTy	$\Delta \vdash \langle\forall a{:}\eta.\tau\rangle@\phi$	$\rightsquigarrow$	$\langle\tau[\phi/a]\rangle$

Push transitivity rules

PushApp	$\Delta \vdash (\gamma_1\ \gamma_2);(\gamma_3\ \gamma_4)$	$\rightsquigarrow$	$(\gamma_1;\gamma_3)\ (\gamma_2;\gamma_4)$	
PushAll	$\Delta \vdash (\forall a{:}\eta.\gamma_1);(\forall a{:}\eta.\gamma_2)$	$\rightsquigarrow$	$\forall a{:}\eta.\gamma_1;\gamma_2$	
PushInst	$\Delta \vdash (\gamma_1@\tau);(\gamma_2@\tau)$	$\rightsquigarrow$	$(\gamma_1;\gamma_2)@\tau$	when $\Delta \vdash^{\text{co}} \gamma_1;\gamma_2 : \sigma_1 \sim_{\#} \sigma_2$
PushNth	$\Delta \vdash (nth\ k\ \gamma_1);(nth\ k\ \gamma_2)$	$\rightsquigarrow$	$nth\ k\ (\gamma_1;\gamma_2)$	when $\Delta \vdash^{\text{co}} \gamma_1;\gamma_2 : \sigma_1 \sim_{\#} \sigma_2$

Figure 7 Coercion simplification (I)

4.1.2 Pushing symmetry down

Uses of symmetry, contrary to reflexivity, are pushed as close to the leaves as possible or eliminated, (rules SymRefl, SymAll, SymApp, SymTrans, and SymSym) only getting stuck at terms of the form $sym\ x$ and $sym\,(C\ \overline{\gamma})$. The idea is that by pushing uses of symmetry towards the leaves, the rest of the rules may completely ignore symmetry, except where symmetry-pushing gets stuck (variables or axiom applications).

4.1.3 Reducing coercions

Rules RedNth, RedInstCo, and RedInstTy comprise the first interesting group of rules. They eliminate uses of injectivity and instantiation. Rule RedNth is concerned with the case where we wish to decompose a coercion of type $H\ \overline{\phi} \sim_{\#} H\ \overline{\sigma}$, where the coercion term contains H in its head. Notice that H is a type and may already be applied to some type arguments $\overline{\tau}^{1..\ell}$, and hence the rule has to account for selection from the first ℓ arguments, or a later argument. Rule RedInstCo deals with instantiation of a polymorphic coercion with a type. Notice that in rule RedInstCo the quantified variable may only appear "protected" under some $\langle\sigma\rangle$ inside γ, and hence simply substituting $\gamma[\phi/a]$ is guaranteed to produce a syntactically well-formed coercion. Rule RedInstTy deals with the instantiation of a polymorphic coercion that is *just* a type.

4.1.4 Eta expanding and subsequent reducing

Redexes of RedNth and RedInstCo or RedInstTy may not be directly visible. Consider $nth\ k\ (\langle H\ \overline{\tau}^{1..\ell}\rangle\ \overline{\gamma}; \gamma)$. The use of transitivity stands in our way for the firing of rule RedNth. To the rescue, rules EtaAllL, EtaAllR, EtaNthL, and EtaNthR, push decomposition or instantiation through transitivity and eliminate such redexes. We call these rules "eta" because in effect we are η-expanding and immediately reducing one of the components of the transitive composition. Here is a decomposition of EtaAllL in smaller steps that involve an η-expansion (of γ_2 in the second line):

$$
\begin{array}{ll}
 & ((\forall a{:}\eta\,.\,\gamma_1); \gamma_2)@\phi \\
\rightsquigarrow & ((\forall a{:}\eta\,.\,\gamma_1); (\forall a{:}\eta\,.\,\gamma_2@a))@\phi \\
\rightsquigarrow & (\forall a{:}\eta\,.\,\gamma_1; \gamma_2@a)@\phi \quad \rightsquigarrow \quad \gamma_1[\phi/a]; \gamma_2@\phi
\end{array}
$$

We have merged these steps in a single rule to facilitate the proof of termination. In doing this, we do not lose any reactions, since all of the intermediate terms can also reduce to the final coercion.

There are many design possibilities for rules that look like our η-rules. For instance one may wonder why we are not always expanding terms of the form $\gamma_1; (\forall a{:}\eta\,.\,\gamma_2)$ to $\forall a{:}\eta\,.\,\gamma_1@a; \gamma_2$, whenever γ_1 is of type $\forall a{:}\eta\,.\,\tau \sim_{\#} \forall a{:}\eta\,.\,\phi$. We experimented with several variations like this, but we found that such expansions either complicated the termination argument, or did not result in smaller coercion terms. Our rules in effect perform η-expansion *only* when there is a firing reduction directly after the expansion.

4.1.5 Pushing transitivity down

Rules PushApp, PushAll, PushNth, and PushInst push uses of transitivity *down* the structure of a coercion term, towards the leaves. These rules aim to reveal more redexes at the leaves, that will be reduced by the next (and final) set of rules. Notice that rules

$$\frac{\Delta \vdash^{co} c : \tau \sim_{\#} \upsilon}{\Delta \vdash c; sym\ c \rightsquigarrow \langle \tau \rangle}\ \textsc{VarSym} \qquad \frac{\Delta \vdash^{co} c : \tau \sim_{\#} \upsilon}{\Delta \vdash sym\ c; c \rightsquigarrow \langle \upsilon \rangle}\ \textsc{SymVar}$$

$$\frac{(C\ \overline{(a{:}\eta)} : \tau \sim_{\#} \upsilon) \in \Gamma \quad \overline{a} \subseteq ftv(\upsilon)}{\Delta \vdash C\ \overline{\gamma}_1; sym\,(C\ \overline{\gamma}_2) \rightsquigarrow [\overline{a} \mapsto \overline{\gamma_1; sym\ \gamma_2}]\uparrow(\tau)}\ \textsc{AxSym}$$

$$\frac{(C\ \overline{(a{:}\eta)} : \tau \sim_{\#} \upsilon) \in \Gamma \quad \overline{a} \subseteq ftv(\tau)}{\Delta \vdash sym\,(C\ \overline{\gamma}_1); C\ \overline{\gamma}_2 \rightsquigarrow [\overline{a} \mapsto \overline{sym\ \gamma_1; \gamma_2}]\uparrow(\upsilon)}\ \textsc{SymAx}$$

$$\frac{(C\ \overline{(a{:}\eta)} : \tau \sim_{\#} \upsilon) \in \Gamma \quad \overline{a} \subseteq ftv(\upsilon) \quad nontriv(\delta) \quad \delta = [\overline{a} \mapsto \overline{\gamma}_2]\uparrow(\upsilon)}{\Delta \vdash (C\ \overline{\gamma}_1); \delta \rightsquigarrow C\ \overline{\gamma_1;\gamma_2}}\ \textsc{AxSuckR}$$

$$\frac{(C\ \overline{(a{:}\eta)} : \tau \sim_{\#} \upsilon) \in \Gamma \quad \overline{a} \subseteq ftv(\tau) \quad nontriv(\delta) \quad \delta = [\overline{a} \mapsto \overline{\gamma}_1]\uparrow(\tau)}{\Delta \vdash \delta; (C\ \overline{\gamma}_2) \rightsquigarrow C\ \overline{\gamma_1;\gamma_2}}\ \textsc{AxSuckL}$$

$$\frac{(C\ \overline{(a{:}\eta)} : \tau \sim_{\#} \upsilon) \in \Gamma \quad \overline{a} \subseteq ftv(\tau) \quad nontriv(\delta) \quad \delta = [\overline{a} \mapsto \overline{\gamma}_2]\uparrow(\tau)}{\Delta \vdash sym\,(C\ \overline{\gamma}_1); \delta \rightsquigarrow sym\,(C\ \overline{sym\ \gamma_2;\gamma_1})}\ \textsc{SymAxSuckR}$$

$$\frac{(C\ \overline{(a{:}\eta)} : \tau \sim_{\#} \upsilon) \in \Gamma \quad \overline{a} \subseteq ftv(\upsilon) \quad nontriv(\delta) \quad \delta = [\overline{a} \mapsto \overline{\gamma}_1]\uparrow(\upsilon)}{\Delta \vdash \delta; sym\,(C\ \overline{\gamma}_2) \rightsquigarrow sym\,(C\ \overline{\gamma_2; sym\ \gamma_1})}\ \textsc{SymAxSuckL}$$

Figure 8 Coercion simplification (II)

PushInst and PushNth impose side conditions on the transitive composition $\gamma_1;\gamma_2$. Without these conditions, the resulting coercion may not be well-formed. Take $\gamma_1 = \forall a{:}\eta.\langle T\ a\ a\rangle$ and $\gamma_2 = \forall a{:}\eta.\langle T\ a\ \texttt{Int}\rangle$. It is certainly the case that $(\gamma_1@\texttt{Int});(\gamma_2@\texttt{Int})$ is well formed. However, $\vdash^{co} \gamma_1 : \forall a{:}\eta.\ T\ a\ a \sim_{\#} \forall a{:}\eta.\ T\ a\ a$ and $\vdash^{co} \gamma_2 : \forall a{:}\eta.\ T\ a\ \texttt{Int} \sim_{\#} \forall a{:}\eta.\ T\ a\ \texttt{Int}$, and hence $(\gamma_1;\gamma_2)@\texttt{Int}$ is not well-formed. A similar argument applies to rule PushNth.

4.1.6 Leaf reactions

When transitivity and symmetry have been pushed as low as possible, new redexes may appear, for which we introduce rules VarSym, SymVar, AxSym, SymAx, AxSuckR, AxSuckL, SymAxSuckR, SymAxSuckL. (Figure 8)

- Rules VarSym and SymVar are entirely straightforward: a coercion variable (or its symmetric coercion) meets its symmetric coercion (or the variable) and the result is the identity.
- Rules AxSym and SymAx are more involved. Assume that the axiom $(C\ (\overline{a{:}\eta}){:}\tau \sim_{\#} \upsilon) \in \Gamma$, and a well-formed coercion of the form: $C\ \overline{\gamma}_1; sym\,(C\ \overline{\gamma}_2)$. Moreover $\Delta \vdash^{co} \overline{\gamma}_1 : \overline{\sigma}_1 \sim_{\#} \overline{\phi}_1$ and $\Delta \vdash^{co} \overline{\gamma}_2 : \overline{\sigma}_2 \sim_{\#} \overline{\phi}_2$. Then we know that $\Delta \vdash^{co} C\ \overline{\gamma}_1; sym\,(C\ \overline{\gamma}_2) : \tau[\overline{\sigma}_1/\overline{a}] \sim_{\#} \tau[\overline{\sigma}_2/\overline{a}]$. Since the composition is well-formed, it must be the case that $\upsilon[\overline{\phi}_1/\overline{a}] = \upsilon[\overline{\phi}_2/\overline{a}]$. If $\overline{a} \subseteq ftv(\upsilon)$ then it must be $\overline{\phi}_1 = \overline{\phi}_2$. Hence, the pointwise composition $\overline{\gamma_1; sym\ \gamma_2}$ is well-formed and of type $\overline{\sigma}_1 \sim_{\#} \overline{\sigma}_2$. Consequently, we may replace the original coercion with the *lifting* of τ over a substitution that maps $\overline{a}$ to $\overline{\gamma_1; sym\ \gamma_2}$: $[\overline{a} \mapsto \overline{\gamma_1; sym\ \gamma_2}]\uparrow(\tau)$.
  What is this lifting operation, of a substitution from type variables to coercions, over a type? Its result is a new coercion, and the definition of the operation is given in Figure 9. The easiest way to understand it is by its effect on a type:

$$\boxed{[a \mapsto \gamma]\uparrow(\tau) = \gamma'}$$

$$\begin{array}{lcl}
[a \mapsto \gamma]\uparrow(a) & = & \gamma \\
[a \mapsto \gamma]\uparrow(b) & = & \langle b \rangle \\
[a \mapsto \gamma]\uparrow(H) & = & \langle H \rangle \\
[a \mapsto \gamma]\uparrow(F) & = & \langle F \rangle \\
[a \mapsto \gamma]\uparrow(\tau_1\ \tau_2) & = & \begin{cases} \langle \phi_1\ \phi_2 \rangle \text{ when } [a \mapsto \gamma]\uparrow(\tau_i) = \langle \phi_i \rangle \\ ([a \mapsto \gamma]\uparrow(\tau_1))\ ([a \mapsto \gamma]\uparrow(\tau_2)) \text{ otherwise} \end{cases} \\
[a \mapsto \gamma]\uparrow(\forall b{:}\eta.\, \tau) & = & \begin{cases} \langle \forall a{:}\eta.\, \phi \rangle \text{ when } [a \mapsto \gamma]\uparrow(\tau) = \langle \phi \rangle \\ \forall b{:}\eta.\, ([a \mapsto \gamma]\uparrow(\tau)) \text{ otherwise} \quad (b \notin \mathit{ftv}(\gamma), b \neq a) \end{cases}
\end{array}$$

Figure 9 Lifting

▶ **Lemma 3** (Lifting). *If* $\Delta, (a{:}\eta) \vdash^{\mathrm{ty}} \tau : \eta$ *and* $\Delta \vdash^{\mathrm{co}} \gamma : \sigma \sim \phi$ *such that* $\Delta \vdash^{\mathrm{ty}} \sigma : \eta$ *and* $\Delta \vdash^{\mathrm{ty}} \phi : \eta$, *then* $\Delta \vdash^{\mathrm{co}} [a \mapsto \gamma]\uparrow(\tau) : \tau[\sigma/a] \sim_{\#} \tau[\phi/a]$

Notice that we have made sure that lifting pulls reflexivity as high as possible in the syntax tree – the only significance of this on-the-fly normalization was that it appeared to simplify the argument we have given for termination of coercion normalization.

Returning to rules AxSym and SymAx, we stress that the side condition is essential for the rule to be sound. Consider the following example:

$$C(a{:}\star) : F\ [a] \sim_{\#} \texttt{Int} \in \Gamma$$

Then $(C\ \langle \texttt{Int} \rangle); \mathit{sym}\,(C\ \langle \texttt{Bool} \rangle)$ is well-formed and of type $F\ [\texttt{Int}] \sim_{\#} F\ [\texttt{Bool}]$, but $\langle F \rangle\ (\langle \texttt{Int} \rangle; \mathit{sym}\,\langle \texttt{Bool} \rangle)$ is not well-formed! Rule SymAx is symmetric and has a similar soundness side condition on the free variables of τ this time.

The rest of the rules deal with the case when an axiom meets a lifted type – the reaction swallows the lifted type inside the axiom application. For instance, here is rule AxSuckR:

$$\frac{(C\ (\overline{a{:}\eta}){:}\tau \sim_{\#} \upsilon) \in \Gamma \quad \overline{a} \subseteq \mathit{ftv}(\upsilon) \quad \mathit{nontriv}(\delta) \quad \delta = [\overline{a} \mapsto \overline{\gamma}_2]\uparrow(\upsilon)}{\Delta \vdash (C\ \overline{\gamma}_1); \delta \rightsquigarrow C\ \overline{\gamma_1;\gamma_2}}\ \text{AxSuckR}$$

This time let us assume that $\Delta \vdash^{\mathrm{co}} \overline{\gamma}_1 : \overline{\sigma}_1 \sim_{\#} \overline{\phi}_1$. Consequently $\Delta \vdash^{\mathrm{co}} C\ \overline{\gamma}_1 : \tau[\overline{\sigma}_1/\overline{a}] \sim_{\#} \upsilon[\overline{\phi}_1/\overline{a}]$. Since $\overline{a} \subseteq \mathit{ftv}(\upsilon)$ it must be that $\Delta \vdash^{\mathrm{co}} \overline{\gamma}_2 : \overline{\phi}_1 \sim_{\#} \overline{\phi}_3$ for some $\overline{\phi}_3$ and we can pointwise compose $\overline{\gamma_1;\gamma_2}$ to get coercions between $\overline{\sigma}_1 \sim_{\#} \overline{\phi}_3$. The resulting coercion $C\ \overline{\gamma_1;\gamma_2}$ is well-formed and of type $\tau[\overline{\sigma}_1/\overline{a}] \sim_{\#} \upsilon[\overline{\phi}_3/\overline{a}]$. Rules AxSuckL, SymAxSuckL, and SymAxSuckR involve a similar reasoning.

The side condition $\mathit{nontriv}(\delta)$ is not restrictive in any way – it merely requires that δ contains some variable c or axiom application. If not, then δ can be converted to reflexivity:

▶ **Lemma 4.** *If* $\vdash^{\mathrm{co}} \delta : \sigma \sim_{\#} \phi$ *and* $\neg\ \mathit{nontriv}(\delta)$, *then* $\delta \longrightarrow^{*} \langle \phi \rangle$.

Reflexivity, when transitively composed with any other coercion, is eliminable via ReflElimL/R or and consequently the side condition is not preventing any reactions from firing. It will, however, be useful in the simplification termination proof in Section 6.

The purpose of rules AxSuckL/R and SymAxSuckL/R is to eliminate intermediate coercions in a big transitive composition chain, to give the opportunity to an axiom to meet

its symmetric version and react with rules AxSym and SymAx. In fact this rule is *precisely* what we need for the impressive simplifications from Section 3. Consider that example again:

$$
\begin{array}{lcll}
\gamma_5 & = & \gamma_2;\gamma_3;\gamma_4 & \\
 & = & sym\,(C_N\ \langle\tau_1\rangle);(\langle N\rangle\ \gamma_1);(C_N\ \langle\tau_2\rangle) & (\text{AxSucL with } \delta := (\langle N\rangle\ \gamma_1)) \\
 & \longrightarrow & sym\,(C_N\ \langle\tau_1\rangle);(C_N\ (\gamma_1;\langle\tau_2\rangle)) & (\text{ReflElimR with } \gamma := \gamma_1, \phi := \tau_2) \\
 & \longrightarrow & sym\,(C_N\ \langle\tau_1\rangle);(C_N\ \gamma_1) & (\text{SymAx}) \\
 & \longrightarrow & \langle\rightarrow\rangle\ (\langle\tau_1\rangle;\gamma_1)\ \langle\texttt{Int}\rangle & (\text{ReflElimL with } \phi := \tau_1, \gamma := \gamma_1) \\
 & \longrightarrow & \langle\rightarrow\rangle\ \gamma_1\ \langle\texttt{Int}\rangle &
\end{array}
$$

Notably, rules AxSuckL/R and SymAxSuckL/R generate axiom applications of the form $C\ \overline{\gamma}$ (with a coercion as argument). In our previous papers, the syntax of axiom applications was $C\ \overline{\tau}$, with *types* as arugments. But we need the additional generality to allow coercions rewriting to proceed without getting stuck.

5 Coercion simplification in GHC

To assess the usefulness of coercion simplification we added it to GHC. For Haskell programs that make no use of GADTs or type families, the effect will be precisely zero, so we took measurements on two bodies of code. First, our regression suite of 151 tests for GADTs and type families; these are all very small programs. Second, the `Data.Accelerate` library that we know makes use of type families [5]. This library consists of 18 modules, containing 8144 lines of code.

We compiled each of these programs with and without coercion simplification, and measured the percentage reduction in size of the coercion terms with simplification enabled. This table shows the minimum, maximum, and aggregate reduction, taken over the 151 tests and 18 modules respectively. The "aggregate reduction" is obtained by combining all the programs in the group (testsuite or `Accelerate`) into one giant "program", and computing the reduction in coercion size.

	Testsuite	Accelerate
Minimum	−97%	−81%
Maximum	+14%	0%
Aggregate	**−58%**	**−69%**

There is a substantial aggregate decrease of 58% in the testsuite and 69% in `Accelerate`, with a massive 97% decrease in special cases. These special cases should not be taken lightly: in one program the types and coercions taken together were five times bigger than the term they decorated; after simplification they were "only" twice as big. The coercion simplifier makes the compiler less vulnerable to falling off a cliff.

Only one program showed an increase in coercion size, of 14%, which turned out to be the effect of this rewrite:

$$sym\,(C;D) \quad \longrightarrow \quad (sym\ D);(sym\ C)$$

Smaller coercion terms make the compiler faster, but the normalization algorithm itself consumes some time. However, the effect on compile time is barely measurable (less than 1%), and we do not present detailed figures.

Of course none of this would matter if coercions were always tiny, so that they took very little space in the first place. And indeed that is often the case. But for programs that make heavy use of type functions, un-optimised coercions can dominate compile time. For

Axiom polynomial

$$
\begin{array}{lcl}
p(sym\ \gamma) & = & p(\gamma) \\
p(C\ \overline{\gamma}) & = & z \cdot \Sigma p(\gamma_i) + z + 1 \\
p(c) & = & 1 \\
p(\gamma_1;\gamma_2) & = & p(\gamma_1) + p(\gamma_2) + p(\gamma_1) \cdot p(\gamma_2) \\
p(\langle\phi\rangle) & = & 0 \\
p(nth\ k\ \gamma) & = & p(\gamma) \\
p(\gamma@\phi) & = & p(\gamma) \\
p(\gamma_1\ \gamma_2) & = & p(\gamma_1) + p(\gamma_2) \\
p(\forall a{:}\eta.\gamma) & = & p(\gamma)
\end{array}
$$

Coercion weight

$$
\begin{array}{lcl}
w(sym\ \gamma) & = & w(\gamma) \\
w(C\ \overline{\gamma}) & = & \Sigma w(\gamma_i) + 1 \\
w(c) & = & 1 \\
w(\gamma_1;\gamma_2) & = & 1 + w(\gamma_1) + w(\gamma_2) \\
w(\langle\phi\rangle) & = & 1 \\
w(nth\ k\ \gamma) & = & 1 + w(\gamma) \\
w(\gamma@\phi) & = & 1 + w(\gamma) \\
w(\gamma_1\ \gamma_2) & = & 1 + w(\gamma_1) + w(\gamma_2) \\
w(\forall a{:}\eta.\gamma) & = & 1 + w(\gamma)
\end{array}
$$

Symmetry weight

$$
\begin{array}{lcl}
sw(sym\ \gamma) & = & w(\gamma) + sw(\gamma) \\
sw(C\ \overline{\gamma}) & = & \Sigma sw(\gamma_i) \\
sw(c) & = & 0 \\
sw(\gamma_1;\gamma_2) & = & sw(\gamma_1) + sw(\gamma_2) \\
sw(\langle\phi\rangle) & = & 0 \\
sw(nth\ k\ \gamma) & = & sw(\gamma) \\
sw(\gamma@\phi) & = & sw(\gamma) \\
sw(\gamma_1\ \gamma_2) & = & sw(\gamma_1) + sw(\gamma_2) \\
sw(\forall a{:}\eta.\gamma) & = & sw(\gamma)
\end{array}
$$

Figure 10 Metrics on coercion terms

example, the `Accelerate` library makes heavy use of type functions. The time and memory consumption of compiling all 21 modules of the library are as follows:

	Compile time	Memory allocated	Max residency
With coercion optimisation	68*s*	31 *Gbyte*	153 *Mbyte*
Without coercion optimisation	291*s*	51 *Gbyte*	2,000 *Mbyte*

As you can see, the practical effects can be extreme; the cliff is very real.

6 Termination and confluence

We have demonstrated the effectiveness of the algorithm in practice, but we must also establish termination. This is important, since it would not be acceptable for a compiler to loop while simplifying a coercion, no matter what axioms are declared by users. Since the rules fire non-deterministically, and some of the rules (such as RedInstCo or AxSym) create potentially larger coercion trees, termination is not obvious.

6.1 Termination

To formalize a termination argument, we introduce several definitions in Figure 10. The *axiom polynomial* of a coercion over a distinguished variable z, $p(\cdot)$, returns a polynomial with natural number coefficients that can be compared to any other polynomial over z. The *coercion weight* of a coercion is defined as the function $w(\cdot)$ and the *symmetry weight* of a coercion is defined with the function $sw(\cdot)$ in Figure 10. Unlike the polynomial and coercion weights of a coercion, $sw(\cdot)$ does take symmetry into account. Finally, we will also use the *number of coercion applications and coercion ∀-introductions*, denoted with $intros(\cdot)$ in what follows.

Our termination argument comprises of the lexicographic left-to-right ordering of:

$\mu(\cdot) = \langle p(\cdot), w(\cdot), \mathit{intros}(\cdot), \mathit{sw}(\cdot) \rangle$

We will show that each of the $\rightsquigarrow$ reductions reduces this tuple. For this to be a valid termination argument for $(\longrightarrow)$ we need two more facts about *each* component measure, namely that (i) $(=)$ and $(<)$ are preserved under arbitrary contexts, and (ii) each component is invariant with respect to the associativity of $(;)$.

▶ **Lemma 5.** *If* $\Delta \vDash^{\text{co}} \gamma_1 : \tau \sim_{\#} \sigma$ *and* $\gamma_1 \cong \gamma_2$ *modulo associativity of* $(;)$*, then* $p(\gamma_1) = p(\gamma_2)$*,* $w(\gamma_1) = w(\gamma_2)$*,* $\mathit{intros}(\gamma_1) = \mathit{intros}(\gamma_2)$*, and* $\mathit{sw}(\gamma_1) = \mathit{sw}(\gamma_2)$*.*

Proof. This is a simple inductive argument, the only interesting case is the case for $p(\cdot)$ where the reader can calculate that $p(\gamma_1; (\gamma_2; \gamma_3)) = p((\gamma_1; \gamma_2); \gamma_3)$ and by induction we are done. ◀

▶ **Lemma 6.** *If* $\Gamma, \Delta \vDash^{\text{co}} \gamma_i : \tau \sim_{\#} \sigma$ *(for* $i = 1, 2$*) and* $p(\gamma_1) < p(\gamma_2)$ *then* $p(\mathcal{G}[\gamma_1]) < p(\mathcal{G}[\gamma_2])$ *for any* $\mathcal{G}$ *with* $\Gamma \vDash^{\text{co}} \mathcal{G}[\gamma_i] : \phi \sim_{\#} \phi'$*. Similarly if we replace* $(<)$ *with* $(=)$*.*

Proof. By induction on the shape of $\mathcal{G}$. The only interesting case is the transitivity case again. Let $\mathcal{G} = \gamma; \mathcal{G}'$. Then $p(\gamma; \mathcal{G}'[\gamma_1]) = p(\gamma) + p(\mathcal{G}'[\gamma_1]) + p(\gamma) \cdot p(\mathcal{G}'[\gamma_1])$ whereas $p(\gamma; \mathcal{G}'[\gamma_2]) = p(\gamma) + p(\mathcal{G}'[\gamma_2]) + p(\gamma) \cdot p(\mathcal{G}'[\gamma_2])$. Now, either $p(\gamma) = 0$, in which case we are done by induction hypothesis for $\mathcal{G}'[\gamma_1]$ and $\mathcal{G}'[\gamma_2]$, or $p(\gamma) \neq 0$ in which case again induction hypothesis gives us the result since we are multiplying $p(\mathcal{G}'[\gamma_1])$ and $p(\mathcal{G}'[\gamma_2])$ by the same polynomial. The interesting "trick" is that the polynomial for transitivity contains both the product of the components *and* their sum (since product alone is not preserved by contexts!). ◀

▶ **Lemma 7.** *If* $\Gamma, \Delta \vDash^{\text{co}} \gamma_i : \tau \sim_{\#} \sigma$ *and* $w(\gamma_1) < w(\gamma_2)$ *then* $w(\mathcal{G}[\gamma_1]) < w(\mathcal{G}[\gamma_2])$ *for any* $\mathcal{G}$ *with* $\Gamma \vDash^{\text{co}} \mathcal{G}[\gamma_i] : \phi \sim_{\#} \phi'$*. Similarly if we replace* $(<)$ *with* $(=)$*.*

▶ **Lemma 8.** *If* $\Gamma, \Delta \vDash^{\text{co}} \gamma_i : \tau \sim_{\#} \sigma$ *and* $\mathit{intros}(\gamma_1) < \mathit{intros}(\gamma_2)$ *then* $\mathit{intros}(\mathcal{G}[\gamma_1]) < \mathit{intros}(\mathcal{G}[\gamma_2])$ *for any* $\mathcal{G}$ *with* $\Gamma \vDash^{\text{co}} \mathcal{G}[\gamma_i] : \phi \sim_{\#} \phi'$*. Similarly if we replace* $(<)$ *with* $(=)$*.*

▶ **Lemma 9.** *If* $\Gamma, \Delta \vDash^{\text{co}} \gamma_i : \tau \sim_{\#} \sigma$*,* $w(\gamma_1) \leq w(\gamma_2)$*, and* $\mathit{sw}(\gamma_1) < \mathit{sw}(\gamma_2)$ *then* $\mathit{sw}(\mathcal{G}[\gamma_1]) < \mathit{sw}(\mathcal{G}[\gamma_2])$ *for any* $\mathcal{G}$ *with* $\Gamma \vDash^{\text{co}} \mathcal{G}[\gamma_i] : \phi \sim_{\#} \phi'$*.*

Proof. The only interesting case is when $\mathcal{G} = \mathit{sym}\ \mathcal{G}'$ and hence we have that $\mathit{sw}(\mathcal{G}[\gamma_1]) = \mathit{sw}(\mathit{sym}\ \mathcal{G}'[\gamma_1]) = w(\mathcal{G}'[\gamma_1]) + \mathit{sw}(\mathcal{G}'[\gamma_1])$. Similarly $\mathit{sw}(\mathcal{G}[\gamma_2]) = w(\mathcal{G}'[\gamma_2]) + \mathit{sw}(\mathcal{G}'[\gamma_2])$. By the precondition for the weights and induction hypothesis we are done. The precondition on the weights is not restrictive, since $w(\cdot)$ has higher precedence than $\mathit{sw}(\cdot)$ inside $\mu(\cdot)$. ◀

The conclusion is the following theorem.

▶ **Theorem 10.** *If* $\gamma \cong \mathcal{G}[\gamma_1]$ *modulo associativity of* $(;)$ *and* $\Delta \vDash^{\text{co}} \gamma_1 : \sigma \sim_{\#} \phi$*, and* $\Delta \vdash \gamma_1 \rightsquigarrow \gamma_2$ *such that* $\mu(\gamma_2) < \mu(\gamma_1)$*, it is the case that* $\mu(\mathcal{G}[\gamma_2]) < \mu(\gamma)$*.*

▶ **Corollary 11.** $(\longrightarrow)$ *terminates on well-formed coercions if each of the* $\rightsquigarrow$ *transitions reduces* $\mu(\cdot)$*.*

Note that often the term rewrite literature requires similar conditions (preservation under contexts and associativity), but also *stability under substitution* (e.g. see [1], Chapter 5). In our setting, variables are essentially treated as constants and this is the reason that we do not rely on stability under substitutions. For instance the rule ReflElimR $\Delta | - \gamma; \langle \phi \rangle \rightsquigarrow \gamma$ is *not* expressed as $\Delta | - c; \langle \phi \rangle \rightsquigarrow c$, as would be customary in a more traditional term-rewrite system presentation.

We finally show that indeed each of the $\rightsquigarrow$ steps reduces $\mu(\cdot)$.

▶ **Theorem 12** (Termination). *If* $\Delta \vDash^{\circ} \gamma_1 : \sigma \sim_{\#} \phi$ *and* $\Delta \vdash \gamma_1 \rightsquigarrow \gamma_2$ *then* $\mu(\gamma_2) < \mu(\gamma_1)$.

Proof. It is easy to see that the reflexivity rules, the symmetry rules, the reduction rules, and the η-rules preserve or reduce the polynomial component $p(\cdot)$. The same is true for the push rules but the proof is slightly more interesting. Let us consider PushApp, and let us write p_i for $p(\gamma_i)$. We have that $p((\gamma_1\ \gamma_2);(\gamma_3\ \gamma_4)) = p_1+p_2+p_3+p_4+p_1p_3+p_2p_3+p_1p_4+p_2p_4$. On the other hand $p((\gamma_1;\gamma_3)\ (\gamma_2;\gamma_4)) = p_1+p_3+p_1p_3+p_2+p_4+p_2p_4$ which is a smaller or equal polynomial than the left-hand side polynomial. Rule PushAll is easier. Rules PushInst and PushNth have exactly the same polynomials on the left-hand and the right-hand side so they are ok. Rules VarSym and SymVar reduce $p(\cdot)$. The interesting bit is with rules AxSym, SymAx, and AxSuckR/L and SymAxSuckR/L. We will only show the cases for AxSym and AxSuckR as the rest of the rules involve very similar calculations:

- Case SymAx. We will use the notational convention $\overline{p}_1$ for $p(\overline{\gamma}_1)$ (a vector of polynomials) and similarly $\overline{p}_2$ for $p(\overline{\gamma}_2)$. Then the left-hand side polynomial is:

$$\begin{aligned}&(z\Sigma\overline{p}_1{+}z{+}1) + (z\Sigma\overline{p}_2{+}z{+}1){+}\\ &\qquad (z\Sigma\overline{p}_1{+}z{+}1)\cdot(z\Sigma\overline{p}_2{+}z{+}1) =\\ &(z^2{+}2z)\Sigma\overline{p}_1 + (z^2{+}2z)\Sigma\overline{p}_2 + z^2\Sigma\overline{p}_1\Sigma\overline{p}_2 + (z^2{+}4z{+}3)\end{aligned}$$

  For the right-hand side polynomial we know that each $\gamma_{1i}; sym\ \gamma_{2i}$ will have polynomial $p_{1i} + p_{2i} + p_{1i}p_{2i}$ and it cannot be repeated inside the lifted type more than a finite number of times (bounded by the maximum number of occurrences of a type variable from $\overline{a}$ in type τ), call it k. Hence the right-hand side polynomial is smaller or equal to:

$$k\Sigma\overline{p}_1 + k\Sigma\overline{p}_2 + k\Sigma(p_{1i}p_{2i}) \leq k\Sigma\overline{p}_1 + k\Sigma\overline{p}_2 + k\Sigma\overline{p}_1\Sigma\overline{p}_2$$

  But that polynomial is strictly smaller than the left-hand side polynomial, hence we are done.
- Case AxSuckR. In this case the left-hand side polynomial is going to be greater or equal to (because of reflexivity inside δ and because some of the $\overline{a}$ variables may appear more than once inside υ it is not exactly equal to) the following:

$$\begin{aligned}&(z\Sigma\overline{p}_1 + z + 1) + \Sigma\overline{p}_2 + (z\Sigma\overline{p}_1 + z + 1)\Sigma\overline{p}2 =\\ &\qquad z\Sigma\overline{p}_1\Sigma\overline{p}_2 + z\Sigma\overline{p}_1 + z\Sigma\overline{p}_2 + 2\Sigma\overline{p}_2 + z + 1\end{aligned}$$

  On the other hand, the right-hand side polynomial is:

$$z\Sigma(p_{1i} + p_{2i} + p_{1i}p_{2i}) + z + 1 \;\leq\; z\Sigma\overline{p}_1 + z\Sigma\overline{p}_2 + z\Sigma\overline{p}_1\Sigma\overline{p}_2 + z + 1$$

  We observe that there is a difference of $2\Sigma\overline{p}_2$, but we know that δ satisfies $nontriv(\delta)$, and consequently there must exist some variable or axiom application inside one of the $\overline{\gamma}_2$. Therefore, $\Sigma\overline{p}_2$ is *non-zero* and the case is finished.

It is the arbitrary copying of coercions $\overline{\gamma}_1$ and $\overline{\gamma}_2$ in rules AxSym and SymAx that prevents simpler measures that only involve summation of coercions for axioms or transitivity. Other reasonable measures such as the height of transitivity uses from the leaves would not be preserved from contexts, due to AxSym again.

So far we've shown that all rules but the axiom rules preserve the polynomials, and the axiom rules reduce them. We next show that in the remaining rules, some other component reduces, lexicographically. Reflexivity rules reduce $w(\cdot)$. Symmetry rules preserve $w(\cdot)$ and $intros(\cdot)$ but reduce $sw(\cdot)$. Reduction rules and η-rules reduce $w(\cdot)$. Rules PushApp and PushAll preserve or reduce $w(\cdot)$ but certainly reduce $intros(\cdot)$. Rules PushInst and PushNth reduce $w(\cdot)$. ◀

We conclude that $(\longrightarrow)$ terminates.

6.2 Confluence

Due to the arbitrary types of axioms and coercion variables in the context, we do not expect confluence to be true. Here is a short example that demonstrates the lack of confluence; assume we have the following in our context:

$$C_1\,(a{:}\star \to \star) : F\ a \sim_{\#} a$$
$$C_2\,(a{:}\star \to \star) : G\ a \sim_{\#} a$$

Consider the coercion:

$$(C_1\ \langle\sigma\rangle); sym\,(C_2\ \langle\sigma\rangle)$$

of type $F\ \sigma \sim_{\#} G\ \sigma$. In one reduction possibility, using rule AxSuckR, we may get

$$C_1\ (sym\,(C_2\ \langle\sigma\rangle))$$

In another possibility, using SymAxSuckL, we may get

$$sym\,(C_2\ (sym\,(C_1\ \langle\sigma\rangle)))$$

Although the two normal forms are different, it is unclear if one of them is "better" than the other.

Despite this drawback, confluence or syntactic characterization of normal forms is, for our purposes, of secondary importance (if possible at all for open coercions in such an under-constrained problem!), since we never reduce coercions for the purpose of comparing their normal forms. That said, we acknowledge that experimental results may vary with respect to the actual evaluation strategy, but we do not expect wild variations.

7 Related and future work

Traditionally, work on proof theory is concerned with proof normalization theorems, namely cut-elimination. Category and proof theory has studied the commutativity of diagrams in *monoidal categories* [12], establishing coherence theorems. In our setting Lemma 4 expresses such a result: any coercion that does not include axioms or free coercion variables is equivalent to reflexivity. More work on proof theory is concerned with cut-elimination theorems – in our setting eliminating transitivity completely is plainly impossible due to the presence of axioms. Recent work on *2-dimensional type theory* [10] provides an equivalence relation on equality proofs (and terms), which suffices to establish that types enjoy canonical forms. Although that work does not provide an algorithm for checking equivalence (this is harder to do because of actual computation embedded with isomorphisms), that definition shares many rules with our normalization algorithm. Finally there is a large literature in associative commutative rewrite systems [7, 2].

To our knowledge, most programming languages literature on coercions is not concerned with coercion simplification but rather with inferring the placement of coercions in source-level programs. Some recent examples are [11] and [17]. A comprehensive study of coercions *and their normalization* in programming languages is that of [8], motivated by coercion placement in a language with *type dynamic*. Henglein's coercion language differs to ours in that (i) coercions there are not symmetric, (ii) do not involve polymorphic axiom schemes and (iii) may have computational significance. Unlike us, Henglein is concerned with characterizations of minimal coercions and confluence, fixes an equational theory of coercions, and

presents a normalization algorithm for that equational theory. In our case, in the absence of a denotational semantics for System FC and its coercions, such an axiomatization would be no more ad-hoc than the algorithm and hence not particularly useful: for instance we could consider adding type-directed equations like $\Delta \vdash \gamma \rightsquigarrow \langle \tau \rangle$ when $\Delta \vdash^{\infty} \gamma : \tau \sim_{\#} \tau$, or other equations that only hold in consistent or confluent axiom sets. It is certainly an interesting direction for future work to determine whether there even exists a maximal syntactic axiomatization of equalities between coercions with respect to some denotational semantics of System FC.

In the space of typed intermediate languages, xMLF[14] is a calculus with coercions that capture *instantiation* instead of equality, and which serves as target for the MLF language. Although the authors are not directly concerned with normalization as part of an intermediate language simplifier, their translation of the graph-based instantiation witnesses does produce xMLF normal proofs.

Finally, another future work direction would be to determine whether we can encode coercions as λ-terms, and derive coercion simplification by normalization in some suitable λ-calculus.

Acknowledgments

Thanks to Tom Schrijvers for early discussions and for contributing a first implementation. We would particularly like to thank Thomas Ströder for his insightful and detailed feedback on our draft.

References

1 Franz Baader and Tobias Nipkow. *Term rewriting and all that.* Cambridge University Press, New York, NY, USA, 1998.

2 Leo Bachmair and David A. Plaisted. Termination orderings for associative-commutative rewriting systems. *J. Symb. Comput.*, 1(4):329–349, December 1985.

3 Manuel M. T. Chakravarty, Gabriele Keller, and Simon Peyton Jones. Associated type synonyms. In *ICFP '05: Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming*, pages 241–253, New York, NY, USA, 2005. ACM.

4 Manuel M. T. Chakravarty, Gabriele Keller, Simon Peyton Jones, and Simon Marlow. Associated types with class. *SIGPLAN Not.*, 40(1):1–13, 2005.

5 Manuel M.T. Chakravarty, Gabriele Keller, Sean Lee, Trevor L. McDonell, and Vinod Grover. Accelerating Haskell array codes with multicore GPUs. In *Proceedings of the sixth workshop on Declarative Aspects of Multicore Programming*, DAMP '11, pages 3–14, New York, NY, USA, 2011. ACM.

6 James Cheney and Ralf Hinze. First-class phantom types. CUCIS TR2003-1901, Cornell University, 2003.

7 Nachum Dershowitz, Jien Hsiang, N. Alan Josephson, and David A. Plaisted. Associative-commutative rewriting. In *Proceedings of the Eighth international joint conference on Artificial intelligence - Volume 2*, IJCAI'83, pages 940–944, San Francisco, CA, USA, 1983. Morgan Kaufmann Publishers Inc.

8 Fritz Henglein. Dynamic typing: syntax and proof theory. *Sci. Comput. Program.*, 22:197–230, June 1994.

9 Oleg Kiselyov, Simon Peyton Jones, and Chung-chieh Shan. Fun with type functions. In Cliff Jones and Bill Roscoe, editors, *Reflections on the work of CAR Hoare.* Springer, 2010.

10 Daniel R. Licata and Robert Harper. Canonicity for 2-dimensional type theory. In *Proceedings of the 39th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, pages 337–348, New York, NY, USA, 2012. ACM.

11 Zhaohui Luo. Coercions in a polymorphic type system. *Mathematical Structures in Computer Science*, 18(4):729–751, 2008.

12 Saunders Mac Lane. *Categories for the Working Mathematician*. Number 5 in Graduate Texts in Mathematics. Springer-Verlag, 1971.

13 Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. Simple unification-based type inference for GADTs. In *ICFP '06: Proceedings of the Eleventh ACM SIGPLAN International Conference on Functional Programming*, pages 50–61, New York, NY, USA, 2006. ACM Press.

14 Didier Rémy and Boris Yakobowski. A Church-style intermediate language for MLF. In Matthias Blume, Naoki Kobayashi, and German Vidal, editors, *Functional and Logic Programming*, volume 6009 of *Lecture Notes in Computer Science*, pages 24–39. Springer Berlin / Heidelberg, 2010.

15 Tim Sheard and Emir Pasalic. Meta-programming with built-in type equality. In *Proc 4th International Workshop on Logical Frameworks and Meta-languages (LFM'04), Cork*, pages 106–124, July 2004.

16 Martin Sulzmann, Manuel M. T. Chakravarty, Simon Peyton Jones, and Kevin Donnelly. System F with type equality coercions. In *TLDI '07: Proceedings of the 2007 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation*, pages 53–66, New York, NY, USA, 2007. ACM.

17 Nikhil Swamy, Michael Hicks, and Gavin M. Bierman. A theory of typed coercions and its applications. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming*, ICFP '09, pages 329–340, New York, NY, USA, 2009. ACM.

18 Dimitrios Vytiniotis, Simon Peyton Jones, Tom Schrijvers, and Martin Sulzmann. Outsidein(x): modular type inference with local assumptions. *Journal of Functional Programming*, 21, 2011.

19 Dimitrios Vytiniotisa, Simon Peyton Jones, and Pedro Magalhaea. Equality proofs and deferred type errors. In *Proceedings of ACM SIGPLAN International Conference on Functional Programming (ICFP '12)*, pages 341–352, 2012.

20 Stephanie Weirich, Dimitrios Vytiniotis, Simon Peyton Jones, and Steve Zdancewic. Generative type abstraction and type-level computation. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, pages 227–240, New York, NY, USA, 2011. ACM.